Informatique 2: Travaux Pratiques

UML et Patrons de conception

Le but de cette séance est d'approfondir la notion d'association, de s'initier aux techniques de modélisation en **UML** et d'implémenter les patrons de conceptions vus en cours.
Temps mentionné (🕓) à titre strictement indicatif.

Exercices de base

Question 1: (🕓 *20 minutes*) Ouvrir le fichier *membre_unil.py* disponible sur moodle et établir le diagramme de classes correspondant.

Question 2: (🕓 *20 minutes*) Ouvrir le fichier *fraction.py* disponible sur moodle et établir le diagramme de séquence lors de l'exécution de la dernière ligne du `main` (*ligne 32 : f3 = f1.plus(f2)*).

Question 3: (🕓 *45 minutes*) ***Patron composite.***
Implémenter le patron *composite* (à deux enfants) avec des structures de données **EnsembleAvecMin** contenant des entiers (`int`) et qui possède une méthode **get_min** pour retourner le plus petit entier qu'elles contiennent. Définir deux classes : l'une stockant les entiers qu'elle contient sous forme de liste d'entiers **EnsembleList**, l'autre sous forme de dictionnaire **EnsembleDict** (les clés sont des chaînes de caractères et les valeurs les entiers qui nous intéressent). Tester avec les instructions données dans l'exemple suivant :

>_Exemple

```
ed = EnsembleDict()
ed.ajouter_cle_valeur('A', 10)
ed.ajouter_cle_valeur('B', 12)
ed.ajouter_cle_valeur('C', -5)

el = EnsembleList()
el.ajouter_valeur(4)
el.ajouter_valeur(-8)

ce = CompositeEnsembleAvecMin(el, ed)
ce2 = el + ce
print(ed.get_min(), el.get_min(), ce.get_min(), ce2.get_min())

ed.ajouter_cle_valeur('D', -10)
print(ed.get_min(), el.get_min(), ce.get_min())
other_el = EnsembleList()
```

Vous devriez obtenir

```
-5 -8 -8 -8
-10 -8 -10
```

Définir la méthode `__add__` pour des objets de types **EnsembleAvecMin**, tel que le résultat de **e1 + e2** retourne un objet composé de *e1* et de *e2*.

Question 4: (🕓 *45 minutes*) ***Patron observer.***

1. Implémenter le patron *observer* en définissant deux classes **Observer** et **Observable** ainsi que leurs classes filles. **Tester !**
2. On veut simuler la situation où des tabloïds suivraient des célébrités afin de publier en fonction des activités de ces dernières. Définir une classe **Tabloid** qui a un attribut **nom** et implémente l'interface **Observer** (la méthode **notify** se contente d'afficher le scoop). Faire en sorte que la classe **Individu** (à récupérer sur Moodle - *Exercices ressources > individu_basique.py*) hérite de la classe **Observable**. Notifier les tabloïds lorsqu'un individu se marie. Créer deux objets **Tabloid** qui observent Robin. **Tester !**

>_Exemple

```
robin = Individu('Scherbatsky', 'Robin')

# Public et Sun veulent être notifié lorsqu'un évènement important survient dans la vie de Robin
    Scherbatsky
public = Tabloid('Public')
sun = Tabloid('Sun')

# Pour rappel, Individu doit hériter d'Observable
# Les magazines Public et Sun doivent suivre Robin Scherbatsky pour pouvoir être notifiés
robin.ajouter_observer(public)
robin.ajouter_observer(sun)

# Lorsque Robin se marie, tous les tabloids qui le suivent sont notifiés
robin.epouse()
```

Vous devriez avoir :

```
[Public] Scoop! 'Je me marie ! (Scherbatsky Robin)'
[Sun] Scoop! 'Je me marie ! (Scherbatsky Robin)'
```

Question 5: (*30 minutes*) Patron singleton. projet

Implémenter le patron *singleton* dans une classe **Singleton**. Un objet **Singleton** contiendra simplement un attribut privé de type dictionnaire et initialisé avec un dictionnaire vide. Vérifier que deux appels à la méthode **get_instance** renvoient bien le même objet (comparer les adresses mémoire ou utiliser l'opérateur de comparaison ==). Lever une exception si on essaie de créer une instance de la classe **Singleton** alors qu'il en existe déjà une.